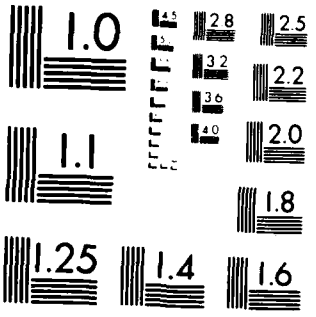
AD-A135 892 SYNTHESIS OF EFFICIENT STRUCTURES FOR CONCURRENT 1/1
COMPUTATION(U) KESTREL INST PALO ALTO CA
R M KING ET AL. 01 OCT 83 KES-U-83-6 AFOSR-TR-83-1060
UNCLASSIFIED F49620-82-C-0007 F/G 9/2 NL

END
DATE FILMED
1 - 84
DTIC



KES.U.83.6

SYNTHESIS of EFFICIENT STRUCTURES for CONCURRENT COMPUTATION

by

Richard M. King

Ernst W. Mayr†

Cordell Green
Principal Investigator

Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304

October 1, 1983

FINAL TECHNICAL REPORT

Prepared for:

Air Force Office of Scientific Research
Building 410
Bolling AFB, DC 20332

Research sponsored by the Air Force Office of Scientific Research (AFSC), United States Air Force, under contract F49620-82-C-0007. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

†cooperating scientist, Stanford University Department of Computer Science, Stanford, CA 94305

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 83-1060	2. GOVT ACCESSION NO. AD-A135892	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SYNTHESIS OF EFFICIENT STRUCTURES FOR CONCURRENT COMPUTATION		5. TYPE OF REPORT & PERIOD COVERED INTERIM, 1 OCT 82-30 SEP 83
		6. PERFORMING ORG. REPORT NUMBER KES.U.83.6
7. AUTHOR(s) Richard M. King and Ernst Mayr		8. CONTRACT OR GRANT NUMBER(s) F49620-82-C-0007
9. PERFORMING ORGANIZATION NAME AND ADDRESS Kestrel Institute 1801 Page Mill Road Palo Alto CA 94304		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Mathematical & Information Sciences Directorate Air Force Office of Scientific Research Bolling AFB DC 20332		12. REPORT DATE 30 SEP 83
		13. NUMBER OF PAGES 18
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Concurrency, parallelism, architectures, synthesis, transformation, interprocessor communication, connectivity reduction, tree structures, systolic arrays.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The object of the research is the development of programming knowledge for the synthesis of concurrent programs. In this interim report techniques are described for synthesizing efficient parallel structures from high level specifications of a problem. These structures contain collections of trees interconnected in various ways. They examine an apparently diverse group of problems and show that they all have properties in common that allow these syntheses to be performed using only a few synthesis rules. They also explore some alternative syntheses for some structures. Some of the synthesis (CONTINUED)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ITEM #20, CONTINUED: paths use transformation rules designed to produce parallel structures containing multidimensional lattices. These lattices are then transformed into structures containing trees in some cases. In other cases the lattice structure is better and is retained. In yet other cases the lattice structure is modified to make a better lattice structure.

Contents

	Page
1 Abstract	1
2 Introduction	1
2.1 Motivation	1
2.2 Outline of TRANSCONS	1
3 Redistributional Problems – Broadcast, Census, Up-and-Down	2
3.1 Broadcasts	3
3.1.1 Bottom-Up Synthesis of a Tree from a Chain	3
3.1.2 Systolic Structure Synthesis	4
3.2 Census Functions	6
4 User-Assisted Aggregation	6
5 Parallel Prefix Computation	8
5.1 Introduction to the Problem	8
5.2 The Divide-and-Conquer Formulation	8
6 Classification Problems	9
6.1 Ordered Classification	10
6.2 Unordered Classification	10
7 String and Pattern Matching	10
8 Conclusions	11
Technical Appendix	12
A.1 Description of the PROCESSORS Statement	12
A.2 Transformations	12
A.3 Creation of a Systolic Structure for Broadcast	14
references	17

Plates

Page

Figure 1. Structure of the Synthesis Process 2
Figure 2. A Bucket Brigade Chain. 2
Figure 3. A Collection Chain 2
Figure 4. Step 1 of a Transformation Into a Balanced Tree 4
Figure 5. Step 1 of a Transformation Into an Unbalanced Tree 4
Figure 6. Simple Parallel Structure for Broadcasting 4
Figure 7. Internal Structure of a Prefix Computation Network 8

§1 Abstract

The object of our research is the development of programming knowledge for the synthesis of concurrent programs. In this final report we describe techniques for synthesizing efficient parallel structures from high level specifications of a problem. These structures contain collections of trees interconnected in various ways. We examine an apparently diverse group of problems and show that they all have properties in common that allow these syntheses to be performed using only a few synthesis rules. We also explore some alternative syntheses for some structures. Some of the synthesis paths use transformation rules designed to produce parallel structures containing multidimensional lattices. These lattices are then transformed into structures containing trees in some cases. In other cases the lattice structure is better and is retained. In yet other cases the lattice structure is modified to make a better lattice structure.

§2 Introduction

The purpose of this report is to explore certain aspects of generating parallel computation structures from high-level specifications. To study this problem we will use a series of classical problems from computer science and show how efficient implementations could be produced semiautomatically or automatically. We are designing TRANSCONS, the TRANSformational CONcurrency Synthesizer, to reduce to practice the principles we enunciate in this report. TRANSCONS will require a theorem prover. Earlier versions of the system will use human assistance to replace or supplement a theorem prover. This assistance takes the form of allowing the system to ask the user to assert a critical theorem or axiom.

§§2.1 Motivation

The system we describe in this report accepts high-level descriptions of a problem and will produces a description of a parallel structure to the topology level. It does not describe the *layouts* of individual processors on a VLSI "chip" or wafer, but the structures produced are simple and regular enough that the problem of producing layouts is tractable.

Parallel structure synthesis can be used in several ways:

- to set up routing in a general purpose parallel computer such as the Ultracomputer [Schwartz-80] or the Universal Parallel Computer [GalPaul-83]
- to design custom VLSI
- to control the configuration phase of Wafer Scale Integration.

In particular, we have studied the feasability of the following:

- "classical" array problems, among them polynomial-time dynamic programming
- systolic array problems such as matrix multiplication, polynomial evaluation, convolution [KungLei-76]
- many graph problems [HMS-83] [LipVal-81]
- search problems such as pattern matching, unification (part of inference), and combinatorial space search.

§§2.2 Outline of TRANSCONS

TRANSCONS is intended to synthesize parallel structures, first semiautomatically and then automatically. The final result is a synthesis system with an efficiency expert [Kant-79] that estimates processor and memory cost as well as time. TRANSCONS asks for little manual assistance (depending on the power of the currently integrated theorem prover).

We have previously explored techniques that produce lattice structures. These intrinsically take linear time (in the extent of one of the dimensions of the problem) to compute their function. We now consider the synthesis of tree structures, in which it is reasonable to hope for solutions to problems in logarithmic time.

In this report we discuss the synthesis of tree structures from structures consisting of chains of processors carrying data in a bucket brigade manner from one end to the other, and from raw specifications using a general divide-and-conquer formulation. We also discuss the synthesis of a certain systolic structure from the chain. Additionally, we describe relationships between parallel implementations of data redistribution problems, prefix summation problems, classification problems, and substring matching. We show how these problems are related and we show uniform techniques for producing good parallel structures for all of these problems.

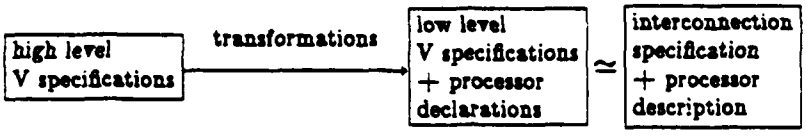


Figure 1. Structure of the Synthesis Process

The programming language V, in which TRANSCONS, its inputs, and its outputs are written, is a wide spectrum language that can specify conceptual schemata, specifications, low-level programs, program transformation rules, and (with some special extensions) processor interconnections. As far as V is used in this report it will be self-explanatory. The **PROCESSORS** statement, used to specify multiple processors and interconnections, will be described in some detail in the Appendix.

§3 Redistributional Problems – Broadcast, Census, Up-and-Down

In this section we consider the class of problems in which data are either available at a central source and where there is an opportunity to simultaneously operate on this data at multiple sites, or where data are available at diverse sources and where it is necessary to summarize this data in a single place. We also consider problems which combine these features.

We start with a high-level specification for a given problem. In the synthesis process we develop a low level specification which is functionally equivalent, but which exhibits concurrency and which describes the programs to be run on the various processors.

Suppose we have a broadcast problem. A naïve solution to this problem is a chain of processors. See Figure 2. In [**King-83**] we studied a formal method to obtain such a configuration from the high level problem specification.

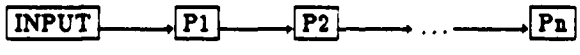


Figure 2. A Bucket Brigade Chain

An analogous technique can produce a chain in the reverse direction (see Figure 3) for computing, for example, the sum over a list. Here the desired value is computed incrementally, and the partial sum is passed from one processor to the next. In each processor a new contribution is added in.

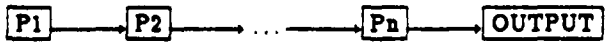
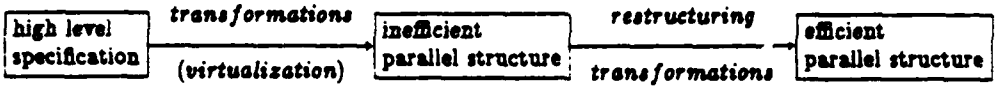


Figure 3. A Collection Chain

These structures are no faster than linear in the size of the problem. We explore ways to improve this.

TRANSCONS can be described in a diagram as follows:



In this paper we do not describe the first step of the above diagram. This was studied in [Kung-83]. We study the restructuring transformations of the second step.

§§3.1 Broadcasts

In this subsection we discuss the problem of deriving an efficient parallel structure for broadcasting. There are three alternative paths: A simple version, the bucket brigade chain of Figure 2, can be improved to a tree; the chain can be improved into a systolic array; and the tree structure can be synthesized by divide and conquer. The first two methods will be described below, and the third will be discussed in a later section.

3.1.1 Bottom-Up Synthesis of a Tree from a Chain

This first synthesis path restructures a bucket brigade chain into a functionally equivalent tree. It does this one level at a time. The arity of the tree is arbitrary, but for clarity we will describe the process of synthesizing binary trees.

Balanced trees can be built from chains in the following manner:

- Introduce a chain of new processors half the length of the old.
- Forge a connection from each element i of the new chain to (a): element $2i$ or (b): elements $2i$ and $2i+1$ of the old.
- Unless they are needed for another purpose, (a): cut all the links of the old chain or (b): cut links between elements $2i+1$ and $2i+2$ of the old chain.
- The first element of the old chain received the information to send down the chain over a link. Cut that link and forge a link from its source to the first node of the new chain.
- Iteratively apply this transformation to the new chain until it consists of a single node.

The formal transformation rule contains parts which guarantee applicability and assert the existence of certain functions between old and new chain indices. This is discussed in detail in the Appendix.

The above outline describes two variants of the transformation, a and b. In one variant, a, all of the links of the original chain are cut, and we achieve a balanced binary tree (see Figure 4). In the other variant, b, only one of the links from the new chain to the old chain is forged, and the link from the linked-to old chain element to its successor is not cut. See Figure 5. In this case we achieve a slightly unbalanced tree, but the depth of the tree is the same. The entire structure takes 25% fewer nodes than the one of Figure 4. To pay for this nodes such as P1 must be able to relay messages as well as compute; formerly they would have only needed to compute.

It is evident that the new parallel structures are functionally equivalent to the old, and that they are faster. Any node reachable from the broadcast source in the old parallel structure is reachable in the new one, and the path length is asymptotically half the length. There is no consideration of a bottleneck here, because we are assuming that the same data is received by all of the leaf processors, so all of the internal nodes receive a single value (or the same set of values) and can duplicate that value for their children.

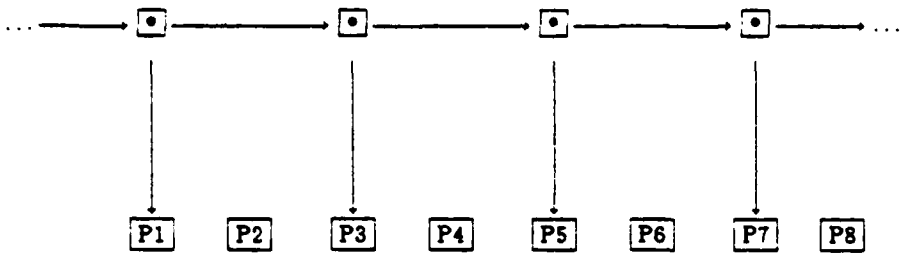


Figure 4. Step 1 of a Transformation Into a Balanced Tree

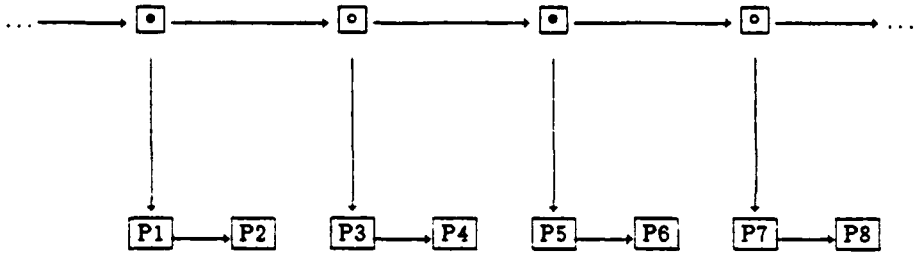


Figure 5. Step 1 of a Transformation Into an Unbalanced Tree

The transformation of Figure 5 leaves corner nodes (marked ∘) buried at all levels of the tree; not merely just above the leaves. These nodes eventually have in-degree=1 and out-degree=1. They can be easily removed, resulting in the 25% saving mentioned above.

3.1.2 Systolic Structure Synthesis

We now study distributional problems preferably implemented by a systolic array. For a specific example, suppose we are evaluating

$\forall i \in \langle\langle 1, \ldots, l \rangle\rangle$; must be enumerated in order
  $\forall j \in \{1, \ldots, n\}$; may be enumerated in any order
    $B_j \leftarrow B_j + A_i$

and suppose that l and n are of the same order of magnitude, or that all of the B-values are in a single place and we choose not to distribute them. A systolic structure is preferable to a tree.

Consider the structure below:

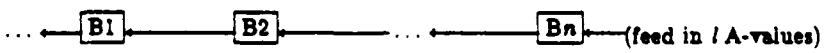
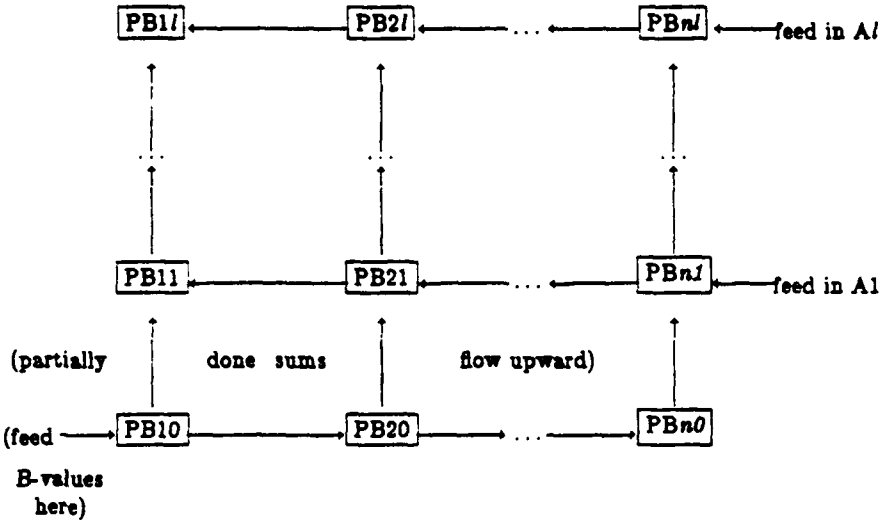


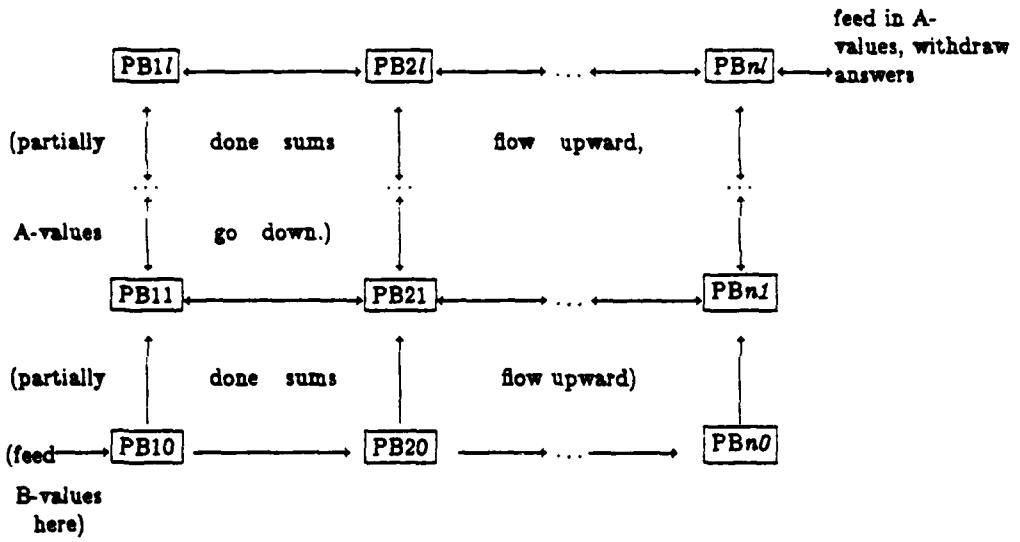
Figure 6. Simple Parallel Structure for Broadcasting

in which each of the l A-values is added to each of the n B-values.

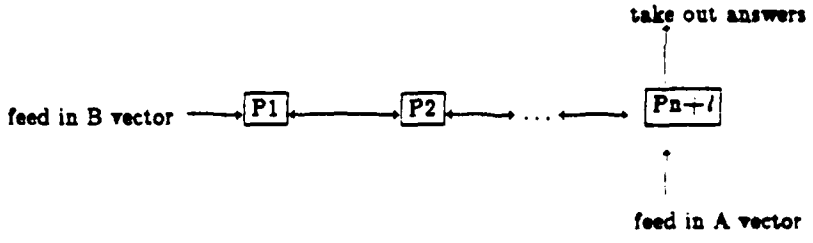
We explicate the l partial sums, using virtualisation. This creates a separate processor for each *step* in the summation process for each of the B-values



and modify that slightly to feed in A-values in only one place.



We then identify P_{ij} with $P_{i+k,j-k}$ for all appropriate k:



This parallel structure is better than the one of Figure 6 because it does not impose synchronous requirements on the I/O capabilities of the system. In Figure 6 it is not shown how B-values get to the various Bi. If this were shown we would see that the assumption was made either that the broadcast problem was embedded in a larger problem that allows the data to already be there or that all n B-processors HEAR the I/O processor. The systolic array shown above allows the I/O processors to be connected to only a single processor.

A formal presentation of these techniques, called virtualization and aggregation, can be found in [King-83].

§§3.2 Census Functions

Trees perform broadcast operations well. They can also be used to compute a class of functions called *Census Functions* (see [LipVal-81]). Examples of census functions include $\sum$, $\prod$, min, and "find the largest subarray of the array that contains a single value". These have in common that they are functions on a string $a_1, a_2, \ldots, a_n$ which can be grouped in an arbitrary manner.

It is possible to perform a census function by using a collection chain (shown in Figure 3). It is always possible to replace such a structure by a tree structure which replaces the task of taking the census operation on a string of length n by the task of taking the census operation on a string of "sums" whose length is $n/2$. The structure of the transformation is the same as the one for broadcasts, making adjustments for the fact that the data have to flow upwards in the tree, and that if a segment of the original chain is allowed to remain than the link tying it to the higher level chain has to be attached to the end of the segment. TRANSCONS contains transformations for census functions as well a broadcasts, but in the interests of brevity these are not presented. The correctness of these transformations is proved in a similar manner to that of the broadcast transformations.

§4 User-Assisted Aggregation

In the previous section we presented methods for generating parallel structures with a tree of communication links. In this section we use the results of that section as a building block as we develop techniques to solve problems such as the normalisation of a set of numbers, standard deviation, and marking the largest interval of elements of a vector that satisfy a given predicate. These problems have in common that information must travel both from a central point to the elements of an array, and *vice versa*.

The naïve method of combining two applications of the techniques of the previous section would yield a *pair* of trees, one with upward communication links and one with downward ones. This uses 50% more nodes than the problem really requires, and it would be desirable to merge these trees in the obvious manner. This involves a new form of aggregation, different from that used to merge diagonal collections of processors in the previous subsection. There, the nodes that were combined had conceptually very similar roles. Here, this will not be the case. We therefore need to merge processors in more general ways than previously.

In examples such as normalization the aggregation is fairly simple to find. In more general cases such as those that arise from graph problems, however, this would not be so easy. Finding aggregations across "family" boundaries can be difficult because of the large number of possibilities involved. Human intervention is necessary here; automation is difficult for the following reasons:

- The bounds of arrays are often arbitrary.
- There are many aggregations available; it is not clear which are useful.
- It is not uncommon for two logical data to share parts of an array.
- It is possible for one array to match the combination of two others.

For these reasons, TRANSCONS understands the **AGGREGATE** statement, of the form:

AGGREGATE $Name_{bound}iters \equiv Pset_{bound}$
HAS $Aset_{bound}$
(**HEARS** $Pname2_{F(bound)}$ $iters2_{bound}$
(**USES** $Aset2_{bound}$ ...))
HAS ...;

This statement is a parameterized statement. The *iters* is a predicate defining permissible bindings of the variables in the list *bound*. It means that those processors in the set $Pset_{bound}$ (which is a set-valued expression) are aggregated (i.e. identified to form a single processor named $Name_{bound}$) for each binding of the variables in *bound* that is permitted by the predicates in *iters*. It is explicitly permitted that the set-valued expression can include enumerated elements and explicit setformers.

The **HAS**, **HEARS** and **USES** elements are analogous to those of a **PROCESSORS** statement (see Appendix), although in an **AGGREGATE** statement there can be more than one **HAS** clause. **HEARS** clauses are associated with specific **HAS** clauses.

When the user provides such assistance searching a potentially enormous set of possible interfamily aggregations is avoided. TRANSCONS's abilities are used to check the validity of the user-proposed aggregation.

The following consistency checking is performed on **AGGREGATE** declarations:

(i) formally specified conditions:

- *Pset* is disjoint for all distinct settings of bound and for all settings of the respective bounds for two **AGGREGATE** statements.
- *Name*((specific bound)) **HAS** (array element) iff $\exists P(F) \in Pset$((specific bound)) that **HAS** (array element).

if $Pname2 = Name$, **then**
$\forall bound$ s.t. *iters*, $bound2 = F(bound)$, $bound \neq bound2$:
$\exists P_{b3} \in PSF\Gamma_{bound}$, $P'_{b4} \in Pset_{bound2}$:
P_{b3} **HEARS** P'_{b4}(**USES** A)

(meaning that the **HEARS** clauses of the **AGGREGATE** statement are those induced by the underlying processors);

(ii) informally specified conditions:

- The order of total amount of computation done by processors underlying a given node does not exceed the length of the longest chain.
- It is not true that A **HEARS** B and B **HEARS** A for the same **USES** datum. (But violation of this condition is likely to imply violation of others)

A simple example of the utility of the **AGGREGATE** statement can be the aggregation of corresponding nodes in two balanced binary trees; one resulting from the identification of a broadcast problem and one from the identification of a census function on the same data.

§5 Parallel Prefix Computation

In this section we will use the divide and conquer scheme to derive a method for performing a parallel prefix computation. To be concrete we use summation in what follows, although the methods described are general.

§§5.1 Introduction to the Problem

Consider the problem of forming the vector of partial sums of the form $\forall i, 0 \leq i \leq n-1: v_i \leftarrow \sum_{j=0}^{i} v_j$ (summation in place). The naïve solution to this problem is (declarations omitted)

$\forall i \in \{0, \ldots, n-1\}$ **do**
  $v'_i \leftarrow \sum_{j \in \{0,\ldots,i\}} v_j$
od

. A slightly less naïve version is

$total \leftarrow 0$
$\forall i \in \langle 0, \ldots, n-1 \rangle$ **do**
  $v'_i \leftarrow total \leftarrow total + v_i$
od

which does allow taking advantage of the cumulative nature of the calculation (i.e. that $\sum_{j=0}^{i} a_j = \sum_{j=0}^{i-1} a_j + a_i$). Probably the best route for deriving the latter from the former is formal differentiation [Paige-79]. The induction variables are v' and a new variable which also receives the value of v'_i in the second line of the first fragment.

We build a structure that binds the vector together using a balanced binary tree. Each of the leaf nodes starts by sending its value to its parent. Each internal node accepts a value from its left son and sends that value to its right son. It also adds the values from its two sons and sends that to its parent. All internal nodes, when they receive a value from their parent, send it to their two children. All leaf nodes add any values received from their parent into their contents. See Figure 7. This structure is best built by a divide-and-conquer scheme.

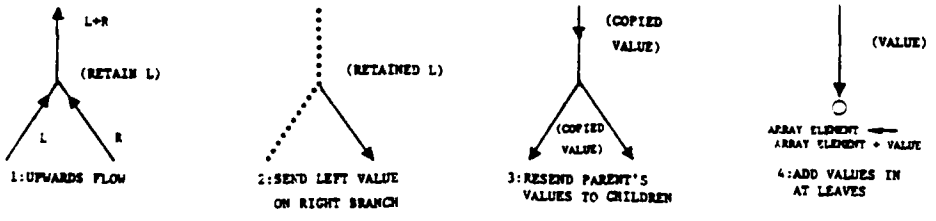


Figure 7. Internal Structure of a Prefix Computation Network

Our model should be distinguished from the "standard" one for parallel prefix circuits (see for example, [Fich-83] and [LadFish-80]) because we permit nodes to be reused and because we require the i^{th} element of the answer to be developed in the same place as the i^{th} element of the input vector. In the cited previous work the circuit was a combinatorial network without memory, i.e. it was required to be a directed acyclic graph and no node would be reused.

§§5.2 The Divide-and-Conquer Formulation

We will derive the tree architecture using a divide and conquer scheme. In what follows, we will use divide and conquer twice. We will specify the unary function $F(\mathbf{z})$ in terms of the divide and conquer scheme and a binary infix operator $s \oplus \mathbf{z}$ which adds s to each element of the vector $\mathbf{z}$.

(The operation is assumed to be addition for concreteness, but the method is general.) We will then specify $\oplus$.

The general binary divide and conquer formulation can be described thus: ($\equiv$ is function definition)

$$F(v) \equiv \begin{cases} \text{if } |v|=1 \text{ then } v \\ \text{otherwise } Combine(F(v'), F(v'')) \end{cases}$$
$$\text{where } v' \parallel v''=v \text{ and } |v'|=\left\lfloor \frac{|v|+1}{2} \right\rfloor$$

which is an instance of the standard divide-and-conquer scheme. It remains to specify the function *Combine*.

In prefix summation the *Combine* operation is concatenation of the two vectors, except that the sum of the elements of the left vector have to be added to the right vector. The sum of the elements in the left vector is always the left vector's last element.

So *Combine* must look like this, to update the right half of the concatenated string:

$$Combine(v, w) \equiv v \parallel v_{|v|} \oplus w$$

where $\oplus$ adds a scalar to each element of a vector. $\oplus$ is, itself, not an atomic operation; it has to be specified. We can define it as follows:

$$s \oplus v \equiv \begin{cases} \text{if } |v|=1 \text{ then } \langle\langle s + v_1 \rangle\rangle \\ \text{otherwise } s \oplus v' \parallel s \oplus v'' \end{cases}$$
$$\text{where } v' \parallel v''=v \text{ and } |v'|=\left\lfloor \frac{|v|+1}{2} \right\rfloor$$

which is itself a divide-and-conquer formulation.

This formulation leads naturally to a tree structure. The time requirement is $2 \lg |v|$ communication times and $\lg |v|$ computation times (additions).

§6 Classification Problems

We are exploring some classifications problems. Specifically, we want fast concurrent solutions to two problems in which there is defined an equivalence relationship on elements of a vector and it is desired to mark a representative of each class induced by the partition implied by the equivalence relationship.

There are two variations of this problem. In one variation, Ordered Classification, there is a total ordering on the equivalence classes and any representative of one class can be compared with a representative of the same or another class quickly. In this case, it is obviously correct to sort the data and find the intervals. In the other variation, Unordered Classification, no such total ordering exists or can be efficiently computed. It is therefore necessary to test the equivalence of every element of the vector directly against every other element. This can still be done swiftly, but it requires more processors to do these comparisons.

It is easy to see* that a total ordering can be imposed whenever equivalence can be tested, but it appears that the cost could be immense†.

---

*The domain of the problem can be Gödel-numbered. In fact, any internal representation scheme of elements of the domain imposes such a numbering. Order elements of the domain by the smallest (under this numbering) element equivalent to a given element under integer comparison (or lexicographic comparison if a variable-length internal representation is being used).

†Suppose the equivalence test costs $\theta(F(l))$ where l is the length of this representation. Clearly $F(l) \geq O(l)$. The minimal representation equivalent to a given item can be found by a search in $O(F(l)2^{l+1})$. It is not obvious how to do better.

§§6.1 Ordered Classification

In this problem the desirable approach is to sort the input vector, find adjacent clumps of equivalent elements, and mark the first element of each clump. There are three significant tasks in this; the discovery that sorting is desirable, the provision of a sorting parallel structure, and the marking of first elements of clumps.

There is a parallel structure that sorts in $O(\log n)$ using $O(n \log n)$ processors, but the constants are high [AKS-83]. There is another parallel structure that sorts in $O(\log^2 n)$ on $O(n)$ processors [Batcher-68]. There is a parallel structure that usually sorts in $O(\log n)$ on $O(n)$ processors, but it has a possibility of failure [ReifVal-82]. (This probability tends to zero rapidly as the constants or the problem size increase.)

It may be that there is a relation $\preceq$ such that $x \preceq y \vee y \preceq x$, that $x \preceq y \wedge y \preceq x \Leftrightarrow x \equiv y$, and that $\preceq$ may appear in the resulting parallel structure. If TRANSCONS is presented such an instance of Classification, it will explore synthesis paths consisting of a sort followed by a prefix computation, and it will also explore a parallel structure similar to the one described in the next subsection. It will try to select the more efficient one, which will be the sort and prefix structure in exactly those cases where there is an efficiently computable well ordering. There are therefore two pieces of knowledge that are part of TRANSCONS:

- the knowledge that a sort should be considered if the data are well ordered, and
- knowledge of several ways to perform a sort, and the tradeoffs involved.

§§6.2 Unordered Classification

When there is no convenient ordering in a classification problem, TRANSCONS explores two alternative parallel structures. One is fast but uses many processors; the other is slower but uses fewer processors.

The first structure is based on Leighton's Mesh of Trees [Leighton-81]. In this arrangement a rectangular set of processors exists and each processor is responsible for comparing an element of the problem array with a (generally) different element of the same array and deciding which nodes it knows to be redundant on that basis.

One of the sets of trees, call it the horizontal set, is responsible for distributing the data properly to rows of nodes. The data are then propagated to the roots of the vertical set of trees, and then to the vertical nodes. The i^{th} element is then in row i and in column i, so element i, j of the rectangular array of processors can determine whether elements i and j are equivalent. This information is propagated up one of the sets of trees and is used to mark the roots of such a set. The processor count is $\theta(n^2)$ and the time is $\theta(\log n)$. Note that this is an example of a combination of the census and broadcast techniques.

A trick is available to reduce the number of processors to $\theta(n^2/\log n)$, clearly the best available in that time because of the number of comparisons that have to be made. Instead of n columns, $n/\log n$ columns can be provided. Each node is responsible for performing $\log n$ comparisons instead of just one, but this only slows the process by a constant factor.

The other structure is slower but uses fewer processors. It can be synthesized by straightforward use of aggregation on the larger structure (where the larger structure uses chains instead of trees).

§7 String and Pattern Matching

The last problem we investigate is the string search problem. In this problem, we are trying to determine the position of the first occurrence of one of its arguments, a string (called the *pattern*), in the second argument, a longer string (called the *text*). TRANSCONS can handle a string matching problem using a double application of a broadcast tree synthesis followed by double application of a census tree synthesis.

It is clear that if the length of the pattern is l and that of the text is m, then the simple method of trying every possible position of the pattern within the text will use ml comparisons.

The base form of the problem is

$$result \leftarrow \min_{i \in \{1,\ldots,|S|-|S'|\}} \forall j \in \{1,\ldots,|S'|\} S_{i+j-1} = S'_j$$

This form requires $O((|S|)(|S'|))$ time. TRANSCONS requires $O((|S|)(|S'|))$ processors to solve the problem in $O(\log((|S|)(|S'|)))$ time.

Replacing a $\forall \ldots$ used as a boolean by an $\wedge_{\ldots}$, we get

$$result \leftarrow \min_{i \in \{1,\ldots,|S|-|S'|\}} \left[\bigwedge_{j \in \{1,\ldots,|S'|\}} S_{i+j-1} = S'_j \right]$$

Virtualization around both the $\wedge_{\ldots}$ and the $\forall \ldots$ yields the two-dimensional array which we will call *equal*.

ARRAY $equal_{i,j}, j \in \{1,\ldots,|S'|\}, i \in \{j,\ldots,|S|-|S'|+j\}$
$\forall j \in \{1,\ldots,|S'|\}, i \in \{j,\ldots,|S|-|S'|+j\}$ **do**
  $equal_{i,j} \leftarrow S_i = S'_j$

$$result \leftarrow \min_{i \in \{1,\ldots,|S|-|S'|\}} \left[\bigwedge_{j \in \{1,\ldots,|S'|\}} equal_{i,j} \right]$$

produces a parallel structure in which chains will be created in two mutually orthogonal directions corresponding to the two dimensions of *equal* (to distribute S and S' characters), and along each 45° diagonal (to collect information for the $\wedge \ldots$ operation).

There will be three collections of trees. One will be "horizontal" and a second "vertical" in the lattice of leaf processors. These sets derive from the distribution problem inherent in this approach to pattern matching. A third collection of trees is diagonal. Its source is one of the census problem represented by the $\wedge$ of the specification. There is another tree connecting the root nodes of the diagonal trees; it derives from the min census function.

§8 Conclusions

In this paper we have explored the problem of communication among a large number of processors when the nature of the problem is such that either large amounts of data must be summarised in a single processor, small amounts of data must be distributed among many processors. We have also explored combinations of these techniques. By design, TRANSCONS is able to combine these techniques and others to produce efficient parallel structures from high-level specifications.

We have also explored some of the efficiency issues that must be considered. A systolic array can be the best parallel structure for one of these problems. Advanced versions of TRANSCONS detect these cases and synthesise that better implementation.

We conclude that the problem of producing efficient parallel structures for the class of problems discussed in this paper is amenable to automation through the use of a transformational system. We are now completing the design of and implementing TRANSCONS, a testbed for these techniques and (hopefully) a practical result when completed.

Technical Appendix

§A.1 Description of the PROCESSORS Statement

The concurrent V language includes a **PROCESSORS** statement to specify concurrency.

Any part of the **PROCESSORS** statement except the processors definition clause can be made conditional (evaluable at "compile time"). The **HAS** clause describes data that a given processor is responsible for. The **HEARS** clause defines "wires" that can be used to receive signals from other processors. The **USES** clauses attached to a given **HEARS** clause define data that are expected to arrive on the corresponding wire. This is useful to detect groups of wires that allow a signal to propagate from one processor to another one that is not directly connected to it, and it helps to define the internal program of that processor by defining an allocation of space in an internal table associated with that wire.

The example below is a **PROCESSORS** statement. It describes a family of processors named P that comprises half of a square array, cut along a main diagonal. Each processor $P_{l,m}$ in this family **HAS** (is responsible for computing) an array element $A_{l,m}$. The processors along one of the edges of the original square **HEAR** (receive a connection from) a processor named Q (which is a family containing one element) because the l^{th} element of that edge **USES** (needs the value of) array element v_l. Similarly, $P_{l,m}$ **HEARS** $P_{l,m-1}$ because it **USES** $A_{l,k}$ for any $1 \leq k \leq m-1$. We call the **USES** clause(s) attached to a **HEARS** clause the *motivation(s)* for that **HEARS** clause.

Observe that the processors as nodes and the **HEARS** clauses as edges constitute a graph. If a processor **HEARS** another processor for a **USE** of a value there must be a path from the processor that **HAS** that value to the processor that **USES** it, and the last edge of that path must be the **HEARS** clause attached to the **USES** clause.

PROCESSORS $P_{l,m}, 1 \leq m \leq n, 1 \leq l \leq n-m+1$ **HAS** $A_{l,m}$
  If $m=1$ **then HEARS** Q **(USES** v_l**)**
  If $2 \leq m \leq n$ **then HEARS** $P_{l,m-1}$ **(USES** $A_{l,k}, 1 \leq k \leq m-1$**)**
  If $2 \leq m \leq n$ **then HEARS** $P_{l+1,m-1}$ **(USES** $A_{l+k,m-k}, 1 \leq k \leq m-1$**);**

§A.2 Transformations

Suppose we start with the following specification of a broadcast problem:

INPUT ARRAY $a_j, j \in \{1, \ldots, l\}$
INPUT ARRAY $b_k, k \in \{1, \ldots, n\}$
OUTPUT ARRAY $c_k, k \in \{1, \ldots, n\}$
ARRAY $internal_k, k \in \{1, \ldots, n\}$
$\forall k \in \{1, \ldots, n\}$ **do**
  $internal_k \leftarrow b_k$
$\forall k \in \{1, \ldots, n\}$ **do**
  $\forall j \in \langle\langle 1, \ldots, l \rangle\rangle$ **do**
    $internal_k \leftarrow internal_k + a_j$
$\forall k \in \{1, \ldots, n\}$ **do**
  $c_k \leftarrow internal_k$

To apply section 3 we have to know:

- that we indeed have a chain, i.e. that there is a first processor, a last processor, and a unique path from the first to the last that includes all of the processors that we claim are in the chain.
- That we have a function, F: *processor indices* $\rightarrow$ *integers*, that linearizes the collection of processors properly. $F(i_0)=0$ where i_0 is the index of the first processor in the chain, and $F(a)=F(b)+1$ if processor a directly **HEARS** processor b in the chain. If there are many coexisting non-overlapping (parallel) chains F must produce multiple linearisations, i.e. F: *processor indices* $\rightarrow$ *integer* $\times$ *vector* so that $F(i_0)=\langle\langle 0, vector \rangle\rangle$ if i_0 is the index of the first processor in *any* chain, $F(a)=\langle\langle i, V \rangle\rangle$ and $F(b)=\langle\langle j, V \rangle\rangle$ is true iff processors a and b are in the same chain, and $F(a)=\langle\langle i, V \rangle\rangle$ and $F(b)=\langle\langle i+1, V \rangle\rangle$ is true iff processor a directly **HEARS** processor b in the chain.
- F has an inverse. (This allows us to compute the processors given their places in named chains.)
- We can enumerate the chains, i.e. we can write an enumeration that gets all V such that $F^{-1}(\langle\langle 0, V \rangle\rangle)$ exists.
- Given V, it's possible to determine how high i gets such that $\langle\langle i, v \rangle\rangle$ is in the range of F for valid processor id's.

In terms of the notation we use for parallel structures, this is performed using the following steps:

- Build a new PROCESSORS statement declaring that family P' mentioned below. It needs to use the domain finding function described in the last group of items. It HAS the values desired by the chain.
- Provide this new PROCESSORS statement with a chain connecting the nodes in linear order. This will certainly be possible; the first part of the processor index exposes the linearity explicitly.
- Cut the chain between n-sized clusters of nodes in the chain, unless there is another need for these links. This can be accomplished in the following manner:
  - The **HEARS** clause of the chain has a **USES** clause or clauses referring to the values that are being passed down the chain. If we need a parallel structure that uses fewer nodes (see discussion below) attach a condition to the uses clause(s) for these values that inhibits them for processors with index a for which $F(a)=\langle\langle i, V \rangle\rangle$ and i is a multiple of n. If a straight balanced tree is acceptable, eliminate the uses clause(s) completely. It is only reasonable to use the "fewer nodes" parallel structure when $n=2$.
  - Fabricate a new **HEARS** clause that hears a processor $P'_{i/n,V}$ for all processors meeting the above condition. P' is a new family. Let this **HEARS** clause have a **USES** clause that uses the value(s) described above.
  - Attach conditions to the chain's **HEARS** clause that inhibits **HEAR**ing if none of the conditions on any of its **USES** clauses is active. (This will eliminate a wire if the only purpose of the chain was its bucket brigade function.)
- The first processor of each chain has to HEAR the same processor that was formerly **HEAR**d by the first processor of the original chain.
- repeat as necessary.

The implementation of this transformation in TRANSCONS rules is as follows:

```
(rule Halve-Chain (**) transform
  **:'PROGRAM
  ∧ Old-Ps:'PROCESSORS PP(bounds) vary iters has AA(bounds2) iters2 ;
    if P(bounds) then HEARS PP(F(bounds))(USES BB(jj));
    ...'
  ∧   Old-Ps ∈ **
  ∧ (THEOREM ((F(A)=F(B)) ⇒ A=B))       ; 1-1
  ∧ (THEOREM ((F(A)=B) ⇒ A=F'(B)))      ; has inverse
  ∧ (THEOREM ((G(i,V)=A) ⇒ G(i + 1,V)=F'(A)))     ; linearizable
  ∧ (THEOREM ((G(i,V)=G(j,W)) ⇒ i=j  ∧ V=W))     ; linearisation is 1-1
  ∧ (THEOREM (P(A) ⇔ ∃V|G(0,V)=A))    ; and starts from 0
  ∧ (THEOREM ((P(G(0,V)) ⇒ P(G(1,V))))     ; the chains each have
                                        ; two nodes
  ∧ (THEOREM (A=G(H(A))))  ; linearization has inverse
  ∧ var ∈ bounds ⇒ ~(FREEIN var jj)   ; different processors all
                                        ; want the same info
  ∧ Newnodes = (GENSYM 'NODE)    ; we'll want to create the new chain
  ∧ New-Ps:'PROCESSORS PP(bounds) vary iters has AA(bounds2) iters2;
    if P(bounds) then HEARS Newnodes(⌊(H(BOUNDS)(1)/2), H(BOUNDS)(2..)⌋)
          (USES BB(jj));
    ...'          ; lets cut the links in the old chain and
                  ; forge links from the new one to the old
                  ; one's nodes
  ∧ Newer-Ps:'PROCESSORS Newnodes(nbounds) vary
        iters[boundsmathrel\G(nbounds)]&ODDP(nbounds(1))
        HASØ;
    if nbounds(1)  >  0
      then HEARS Newnodes(nbounds(1)—2, nbounds(2:....))
          (USES BB(jj))'    ; and build the new chain
  →
  Old-Ps ∉ **
  ∧   New-Ps ∈ **
  ∧   Newer-Ps ∈ **
)
```

The conditions on correctness are checked by the **THEOREM** assertions.

The other option discussed in that section, the use of the next-to-the-leaves nodes to pass data and compute simultaneously, requires different rules, shown below:

Rules for THIS change have been omitted for brevity.

§A.3 Creation of a Systolic Structure for Broadcast

We can synthesise a systolic parallel structure from the base form of the broadcast specification by assigning a single processor to each recipient of the broadcast, for each such processor assigning a column of processors so one is available for each stage of the broadcasting, and then combining diagonal sets of processors into new, single processors in a manner that will be detailed below. We start with:

INPUT ARRAY $a_j, j \in \{1, \ldots, l\}$
INPUT ARRAY $b_k, k \in \{1, \ldots, n\}$
OUTPUT ARRAY $c_k, k \in \{1, \ldots, n\}$
ARRAY $internal_k, k \in \{1, \ldots, n\}$
$\forall k \in \{1, \ldots, n\}$ **do**

$internal_k \leftarrow b_k$
$\forall k \in \{1, \ldots, n\}$ do
  $\forall j \in (\{1, \ldots, l\})$ do
    $internal_k \leftarrow internal_k + a_j$
$\forall k \in \{1, \ldots, n\}$ do
  $c_k \leftarrow internal_k$

as the virtualisation step, we perform a processor expansion, create a chain along the k axis of C' using rule A6, and we have:

INPUT ARRAY $a_j, j \in \{1, \ldots, l\}$
PROCESSORS $PA \ldots$
INPUT ARRAY $b_k, k \in \{1, \ldots, n\}$
PROCESSORS $PB \ldots$
OUTPUT ARRAY $c_k, k \in \{1, \ldots, n\}$
PROCESSORS $PC \ldots$
PROCESSORS $Pinternal'_{k,v} k \in \{1, \ldots, n\}, v \in \{0, \ldots, l\}$ **HAS** $internal'_{k,v}$
  if $k < n$ **then HEARS** $Pinternal'_{k+1,v}$(**USES** $a_{,,}$
  if $k{=}n \wedge v < l$ **then HEARS** $Pinternal'_{k,v+1}$(**USES** a_l)
  if $k{=}n \wedge v{=}l$ **then HEARS** PA(**USES** a_l)
  if $v{=}0 \wedge k > 1$ **then HEARS** $Pinternal'_{k-1,0}$(**USES** b_k)
  if $v{=}0 \wedge k{=}1$ **then HEARS** PB(**USES** b_1)
  if $v > 0$ **then HEARS** $Pinternal'_{k,v-1}$(**USES** $internal'_{k,v-1}$)
ARRAY $internal'_{k,v} k \in \{1, \ldots, n\}, v \in \{0, \ldots, l\}$
(**include** *in* $Pinternal'_{k,0}$:)
  $internal'_{k,0} \leftarrow b_k$
(**if** $j > 0$ **include** *in* $Pinternal'_{k,j}$:)
  $internal'_{k,j} \leftarrow internal'_{k,j-1} + a_j$
(**include** *in* $Pinternal'_{k,l}$:)
  $c_k \leftarrow internal'_{k,l}$

We applied certain techniques of [King-83] once more than necessary to achieve the "countercurrent" effect in which the values flow in opposite directions. This is necessary to make the virtualisation and aggregation work. It also shows the importance of user guidance.

We then aggregate by identifying $Pinternal'_{k,j} \equiv Pinternal'_{k+i,j-i}$ where both exist.

We get

INPUT ARRAY $a_j, j \in \{1, \ldots, l\}$
PROCESSORS $PA \ldots$
INPUT ARRAY $b_k, k \in \{1, \ldots, n\}$
PROCESSORS $PB \ldots$
OUTPUT ARRAY $c_k, k \in \{1, \ldots, n\}$
PROCESSORS $PC \ldots$
PROCESSORS $Pinternal''_{k'}, \exists v \in \{0, \ldots, l\} | k' = v + k \; \forall k \in \{1, \ldots, n\}$
  HAS $internal'_{k,v}, k \in \{1, \ldots, n\},$
    $v \in \{0, \ldots, n\},$
    $k' = k + v$
  if $k' < n + l$ **then HEARS** $Pinternal''_{k'+1}$(**USES** $a_v, v \in \{1, \ldots, l\}$)
  if $k' > -l$ **then HEARS** $Pinternal''_{k'-1}$,(**USES** $b_k, k \in \{1, \ldots, n\}$)
    (**USES** $internal'_{k,v}$
      $k \in \{1, \ldots, n\},$
      $v \in \{0, \ldots, n\},$
      $k' - 1 = k + v$
ARRAY $internal'_{k,v} k \in \{1, \ldots, n\}, v \in \{0, \ldots, l\}$
(**include** *in* $Pinternal''_{k'} | k' \in \{1, \ldots, k\}$:)

$internal'_{k',0} \leftarrow b_{k'}$,
(include in $Pinternal''_{k'}, |k'=k + v|k \in \{1, \ldots, n\}, v \in \{1, \ldots, l\}$
$internal'_{k,j}, \leftarrow internal'_{k,j-1}, +a_j$
(include in $Pinternal''_{k'}|k'=k + l|k \in \{1, \ldots, n\}$:)
$c_k \leftarrow internal'_{k,l}$

. This is the systolic array given in the section.

references

[AKS-83] M. Ajtai, J. Komlós and E. Szemerédi "An $O(n \log n)$ Sorting Network" *Proceedings of the 15th ACM Symposium on Theory of Computing, pp. 1-9*, 1983

[Batcher-68] K. Batcher "Sorting Networks and their Applications" *AFIPS Spring Joint Computer Conference, pp. 307-314*, 1968

[Fich-83] Faith E. Fich "New Bounds for Parallel Prefix Circuits" *Proceedings of the 15th ACM Symposium on Theory of Computing, pp. 100-109*, 1983

[GalPaul-83] Z. Galil and W. Paul "An Efficient General-Purpose Parallel Computer" *Journal of the ACM, vol. 30 #2, pp. 360-387*, April 1983

[HMS-83] P. Hochschild, E. W. Mayr, and A. Siegel "Techniques for Solving Graph Problems in Parallel Environments" *Proceedings of the 24th Symposium on Foundations of Computer Science* to appear November 1983

[Kant-79] Elaine Kant "Efficiency Considerations in Program Synthesis: A Knowledge-Based Approach" , *Ph. D. Thesis, Department of Computer Science, Stanford University*, 1979

[King-83] R. King "Research on Synthesis of Concurrent Computing Systems" *Proceedings of the 10th Symposium on Computer Architecture, pp. 39-46*, 1983

[KingBrown-83] R. King and T. Brown "Proposal for Research On Automatic Synthesis of Tree-Structured Concurrent Computing Systems" , *Kestrel Tech Report #KES.L.83.1*, 1983

[KungLei-76] H. T. Kung and Charles E. Leiserson "Systolic Arrays for VLSI" *Sparse Matrix Proceedings, 1978*

[LadFish-80] R. Ladner and M. Fischer "Parallel Prefix Computation" *Journal of the ACM, vol. 27 #4, pp. 831-838*, 1980

[Leighton-81] F. T. Leighton "A Layout Strategy for VLSI Which is Provably Good" *Proceedings of the 14th ACM Symposium on Theory of Computing, pp. 85-97*, 1982

[LipVal-81] R. J. Lipton and J. Valdes "Census Functions: an Approach to VLSI Upper Bounds" , *Proceedings of the 21st IEEE Symposium on the Foundations of Computer Science, pp. 13-22*, 1981

[Paige-79] R. Paige "Expression Continuity and the Formal Differentiation of Algorithms" *Technical Report #15, Courant Institute, New York, pp. 269-658*, 1979

[ReifVal-82] J. Reif and L. Valiant "A Logarithmic Time Sort for Linear Size Networks" , *Harvard Tech Report #TR-13-82*, 1982

[Schwartz-80] J. Schwartz "Ultracomputers" *ACM TOPLAS, vol. 2 #4 pp. 484-521*, October 1980